Numeric Domains Meet Algebraic Data Types

Santiago Bautista, Thomas Jensen, Benoît Montagu

Numeric Domains Meet Algebraic Data Types

Santiago Bautista
ENS Rennes
Bruz, France
Inria
Rennes, France
santiago.bautista@ens-rennes.fr

Thomas Jensen
Inria
Rennes, France
thomas.jensen@inria.fr

Benoît Montagu
Inria
Rennes, France
benoit.montagu@inria.fr

Abstract

We report on the design and formalization of a novel abstract domain, called *numeric path relations* (NPRs), that combines numeric relational domains with algebraic data types. This domain expresses relations between algebraic values that can contain scalar data. The construction of the domain is parameterized by the choice of a relational domain on scalar values. The construction employs projection paths on algebraic values, and in particular projections on variant cases, whose sound treatment is subtle due to mutual exclusiveness.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Keywords: numeric relations, algebraic types, NPR abstract domain, static analysis, abstract interpretation

ACM Reference Format:
Santiago Bautista, Thomas Jensen, and Benoît Montagu. 2020. Numeric Domains Meet Algebraic Data Types. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3427762.3430178

1 Introduction

The field of relational abstract domains on scalar values is a vibrant research area, that has produced a rich variety of expressive domains [4, 6, 9, 10, 12]. They were successfully used in the implementation of static analyzers to automatically prove the safety of large code bases [3, 5].

In the functional programming community, types are exerted as a means to express properties of structured data—also known as algebraic data types—that a program manipulates. More recently, refinement types [15–17] were developed, that augment types with rich properties, including properties on scalars. Systems based on refinement types use SMT solvers to discharge verification constraints.

The topic of abstract domains for the relational analysis of algebraic data types, however, has remained largely unexplored by the static analysis community. The correlation abstract domain [1] is a recent progress in that area, focused on extracting equality relations. The current work devises a relational abstract domain for algebraic data types, that expresses relations richer than equality over scalar values.

We introduce the novel abstract domain of *Numeric Path Relations* (NPRs) that leverages the expressiveness of known numeric relational domains to denote expressive relations over structured data that contain scalar values. This is a first and necessary step to analyze languages that expose algebraic data types and pattern matching (OCaml, F♯, Swift, Rust...). Such languages have been successfully used to specify and reason on high level models of complex programs, such as operating systems [8]. Our long term goal is to exploit results of static analyses to reduce the manual proof effort required to verify such complex programs.

2 Technical Setting

This section introduces notations for algebraic data types and numeric abstract domains, used in the paper.

2.1 Algebraic Data Types

Algebraic types are arbitrary nestings of sum types and product types. Here, we only deal with non-recursive algebraic types, which excludes lists or trees. Algebraic types are given by the following grammar (N is the type of scalar values):

$$\tau \quad ::= \quad \mathrm{N} \quad | \quad \{\overline{f_i \to \tau_i}^{i\in I}\} \quad | \quad [\overline{A_i \to \tau_i}^{i\in I}]$$

The type $\{\overline{f_i \to \tau_i}^{i\in I}\}$ is the product type where each field f_i is of type τ_i, and $[\overline{A_i \to \tau_i}^{i\in I}]$ stands for the sum type where each constructor A_i takes an argument of type τ_i. For example, playing cards can be represented by the type

$$\left\{ \text{suit} \to \begin{bmatrix} \text{Hearts} \to \{\}, & \text{Spades} \to \{\} \\ \text{Diamonds} \to \{\}, & \text{Clubs} \to \{\} \end{bmatrix}, \ \text{rank} \to \mathrm{N} \right\}$$

Values, written v, are standard for a functional language. From base scalar values $\underline{n}$, compound values are built to form records $\{\overline{f_i \to v_i}^{i\in I}\}$ or variants $A(v)$. The different constructors of a sum type are mutually exclusive: each variant uses

NSAD '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8187-1/20/11…$15.00
https://doi.org/10.1145/3427762.3430178

only *one* constructor. The syntax for values is given by:

$$v \quad ::= \quad \underline{n} \quad | \quad \{\overline{f_i \to v_i}^{i \in I}\} \quad | \quad A(v)$$

For example, let us call hearts_queen the value $\{\text{suit} \to \text{Hearts}(\{\}), \text{rank} \to \underline{12}\}$ that represents the queen of hearts.

We may write *algebraic values* to refer to values from an algebraic data type.

We use projection paths to refer to scalar data contained in an algebraic value. The set of paths, written Paths, is inductively defined by the following grammar:

$$p \quad ::= \quad \varepsilon \quad | \quad .fp \quad | \quad @Ap$$

where f is a field, and A is a constructor.

The projection of an algebraic value v on a path p, written $v \Downarrow^{\text{val}} p$, computes the scalar obtained by following path p in v. It is defined as the following partial function:

$$\begin{aligned} v \Downarrow^{\text{val}} \varepsilon &= v && \text{if } \vdash v : \text{N} \\ \{\overline{f_i \to v_i}^{i \in I}\} \Downarrow^{\text{val}} .f_j p &= v_j \Downarrow^{\text{val}} p && \text{if } j \in I \\ A(v) \Downarrow^{\text{val}} @Ap &= v \Downarrow^{\text{val}} p \end{aligned}$$

E.g., by projecting hearts_queen on its rank field, we get $\underline{12}$: hearts_queen $\Downarrow^{\text{val}}$ .rank = $\underline{12}$. Its projection on the path .suit@Clubs is not defined, however, because the field suit of hearts_queen is not set to the variant Clubs. The projection of hearts_queen over .suit@Hearts is not defined either, as it would lead to the value $\{\}$, which is not a scalar.

2.2 Numeric Abstract Domains

Numeric abstract domains are parameterized by a set $\mathcal{V}$ of variable names. The abstract values of a numeric abstract domain $D(\mathcal{V})$ are pairs (c, V) where $V \subseteq \mathcal{V}$ is a set of variables and c is a set of constraints over the variables of V. For example, the abstract value $(\{x \leq y, x \leq 5\}, \{x, y, z\})$ talks about three variables (x, y and z) and contains two constraints ($x \leq y$ and $x \leq 5$). The variable z is left unconstrained. We call Constraints $(D(\mathcal{V}))$ the set of constraints expressible in the domain $D(\mathcal{V})$. Then, the abstract values of $D(\mathcal{V})$ are elements of $\mathcal{P}(\text{Constraints}(D(\mathcal{V}))) \times \mathcal{P}(\mathcal{V})$ (where $\mathcal{P}$ stands for the powerset). When writing an abstract value (c, V) in examples, we omit V if it corresponds exactly to the variables appearing in the constraints of c. For example, we write $\{x \leq y, x \leq 5\}$ instead of $(\{x \leq y, x \leq 5\}, \{x, y\})$.

A numeric abstract domain $D(\mathcal{V})$ is a tuple

$$\left(\mathcal{A}^{D(\mathcal{V})}, \gamma^{\text{D}(\mathcal{V})}, \sqsubseteq^{D(\mathcal{V})}, \cup^{D(\mathcal{V})}, \cap^{D(\mathcal{V})}\right)$$

$\mathcal{A}^{D(\mathcal{V})}$ is the set of abstract values of $D(\mathcal{V})$. $\gamma^{\text{D}(\mathcal{V})}$ is the concretization function, that maps each abstract value (c, V) to a set of environments, *i.e.* an element of $\mathcal{P}(V \to \mathbf{R})$. The relation $\sqsubseteq^{D(\mathcal{V})}$ is a pre-order on abstract values. $\cup^{D(\mathcal{V})}$ and $\cap^{D(\mathcal{V})}$ are operators on abstract values over-approximating set union and set intersection, respectively. A numeric abstract domain must satisfy the following properties:

- Soundness of pre-order: if $(c, V) \sqsubseteq^{D(\mathcal{V})} (c', V)$, then $\gamma^{\text{D}(\mathcal{V})}((c, V)) \subseteq \gamma^{\text{D}(\mathcal{V})}((c', V))$.
- Soundness of union:

$$\gamma^{\text{D}(\mathcal{V})}(c, V) \cup \gamma^{\text{D}(\mathcal{V})}(c', V) \subseteq \gamma^{\text{D}(\mathcal{V})}\left((c, V) \cup^{D(\mathcal{V})} (c', V)\right)$$

- Soundness of intersection:

$$\gamma^{\text{D}(\mathcal{V})}(c, V) \cap \gamma^{\text{D}(\mathcal{V})}(c', V) \subseteq \gamma^{\text{D}(\mathcal{V})}\left((c, V) \cap^{D(\mathcal{V})} (c', V)\right)$$

These are unsurprising requirements for an abstract domain.

2.3 Variable Manipulation

The concretization, pre-order, abstract union and abstract intersection operations are fundamental to the abstract interpretation framework. Additionally, we assume that the abstract domain D also provides the operations Add, Remove and Rename, for respectively adding new variables to an abstract value, removing some variables, and renaming variables. Those operations are available in Apron [7] and will prove useful in sections 3 and 5.

2.3.1 Adding Variables to an Abstract Value.

The abstract value $\text{Add}_{V_2}(c, V_1)$ *adds* the variables $V_2 \subseteq \mathcal{V}$ to the abstract value (c, V_1). The variables of $\text{Add}_{V_2}(c, V_1)$ are $V_1 \cup V_2$. $\text{Add}_{V_2}(c, V_1)$ should satisfy the following property:

$$\gamma^{\text{D}(\mathcal{V})}\left(\text{Add}_{V_2}(c, V_1)\right) = \left\{ g : (V_1 \cup V_2) \to \mathbf{R} \;\middle|\; \begin{array}{l} \exists f \in \gamma^{\text{D}(\mathcal{V})}((c, V_1)), \\ \forall x \in V_1, f(x) = g(x) \end{array} \right\}$$

Said otherwise, the newly added variables, *i.e.* those in $V_2 \setminus V_1$ are unconstrained in $\text{Add}_{V_2}(c, V_1)$, whereas the variables in V_1 are constrained in the same way as they were in (c, V_1).

2.3.2 Removal of Variables.

$\text{Remove}_{V_2}(c, V_1)$ is obtained by *removing* the variables in $V_2 \subseteq \mathcal{V}$ from (c, V_1). Thus, $\text{Remove}_{V_2}(c, V_1)$ talks about variables in $V_1 \setminus V_2$. In addition, $\text{Remove}_{V_2}(c, V_1)$ should satisfy the following property:

$$\left\{ g : (V_1 \setminus V_2) \to \mathbf{R} \;\middle|\; \begin{array}{l} \exists f \in \gamma^{\text{D}(\mathcal{V})}((c, V_1)), \\ \forall x \in V_1 \setminus V_2, f(x) = g(x) \end{array} \right\} \subseteq \gamma^{\text{D}(\mathcal{V})}\left(\underset{V_2}{\text{Remove}}(c, V_1)\right)$$

In other words, for any environment belonging to the concretization of (c, V_1), its restriction to $V_1 \setminus V_2$ belongs to the concretization of $\text{Remove}_{V_2}(V_1)$. We only require inclusion, not equality. Indeed, removing variables from an abstract value might cause a loss of information in some domains.

2.3.3 Variable Renaming.

Given a bijective function $r : V_1 \to V_2$, where $V_1, V_2 \subseteq \mathcal{V}$; $\text{Rename}_r(c, V_1)$ is obtained by renaming the variables of (c, V_1) into variables of V_2 according to the bijection r. The following property must hold:

$$\gamma^{\text{D}(\mathcal{V})}\left(\underset{r}{\text{Rename}}(c, V_1)\right) = \left\{ g \mid g \circ r \in \gamma^{\text{D}(\mathcal{V})}((c, V_1)) \right\}$$

For example, for $V_1 = \{x, y\}$, $V_2 = \{y, z\}$ and the bijection $r : V_1 \to V_2$ such that $r(x) = y$ and $r(y) = z$, we have:

$$\gamma^{\mathrm{D}(\mathcal{V})}\left(\underset{r}{\mathrm{Rename}}\{x = 2y, y \geq 0\}\right) = \gamma^{\mathrm{D}(\mathcal{V})}\left(\{y = 2z, z \geq 0\}\right)$$

When the domain of r can be deduced from the context and some elements of the domain are left unchanged, we only write the elements modified by r. For example:

$$\gamma^{\mathrm{D}(\mathcal{V})}\left(\underset{[x\mapsto z]}{\mathrm{Rename}}\{x = 2y, y \geq 0\}\right) = \gamma^{\mathrm{D}(\mathcal{V})}\left(\{z = 2y, y \geq 0\}\right)$$

3 Numeric Path Relations

We propose a way to extend any numeric abstract domain D from numeric variables to variables of algebraic data types. The key intuition is to add more structure to variable names using paths: projecting structured data along paths allows to get numeric values that can be abstracted using D. The main challenge is to treat sum types in a sound way. Indeed, projecting a value of a sum type conveys implicit information about which constructor was used to build that value.

Definition 3.1 (Extended variable)**.** An *extended variable* is a pair of a variable and a path. We call $\mathcal{S} = \mathcal{V} \times \mathrm{Paths}$ the set of all extended variables.

Definition 3.2 (Numeric Path Relation)**.** We call *numeric path relation* (NPR) any abstract value from a numeric domain $D(\mathcal{S})$ on extended variables, *i.e.*, any pair (c, P) of $\mathcal{A}^{D(\mathcal{S})} = \mathcal{P}(\mathrm{Constraints}\,(D(\mathcal{S}))) \times \mathcal{P}(\mathcal{S})$.

In the rest of this section, we define γ^{NPR}, $\sqsubseteq^{\mathrm{NPR}}$, $\cup^{\mathrm{NPR}}$ and $\cap^{\mathrm{NPR}}$, that confer to NPRs the structure of relational abstract domain over values of algebraic data types.

To design a static analysis, we need to abstractly represent sets of states, denoting the possible environments from variables to values, that are reachable at a given program point. To do this, we exploit projection on paths, that allow to go from algebraic values to scalar values.

Given an environment E that maps variables from a set V to algebraic values, and given a set $P \in \mathcal{S}$ of variable-path pairs, we define the projection of E over P as the function

$$\begin{array}{rlll} E \Downarrow^{\mathrm{env}} P : & P & \to & \mathbf{R} \\ & (x, p) & \mapsto & E(x) \Downarrow^{\mathrm{val}} p \end{array}$$

The function $E \Downarrow^{\mathrm{env}} P$ can be seen as a numeric environment on extended variables. Notice that $E \Downarrow^{\mathrm{env}} P$ is defined only if $E(x) \Downarrow^{\mathrm{val}} p$ is defined for every $(x, p) \in P$. For example, for the value hearts_queen from Section 2.1, $[x \mapsto \mathrm{hearts\_queen}] \Downarrow^{\mathrm{env}} \{(x, .\mathrm{suit@Clubs})\}$ is *not* well defined. In particular, if there are paths in P that try to project the same value over *different* constructors of a sum type, then $E \Downarrow^{\mathrm{env}} P$ is never well-defined, independently of the environment E. *E.g.*, for $P = \{(x, .\mathrm{suit@Clubs}), (x, .\mathrm{suit@Hearts})\}$, there is no E such that $E \Downarrow^{\mathrm{env}} P$ is well-defined. In such a case we say that P is *inconsistent*.

```
drift (x : t1) = match x with
  A(n) → r := A(n+1)
| B(n) → r := B(n-1)
```

Figure 1. The drift function

Definition 3.3 (Concretization of NPRs)**.** We define the concretization function on NPRs by:

$$\gamma^{\mathrm{NPR}}\left((c, P)\right) = \left\{ E \;\middle|\; \begin{array}{ll} E \Downarrow^{\mathrm{env}} P \text{ is well defined} & \wedge \\ E \Downarrow^{\mathrm{env}} P \in \gamma^{\mathrm{D}(\mathcal{S})}\left((c, P)\right) & \end{array} \right\}$$

Notice that for any inconsistent P (*e.g.* $\{(x, @A), (x, @B)\}$), we have $\gamma^{\mathrm{NPR}}\left((c, P)\right) = \emptyset$, no matter the constraint set c.

Consider for example the drift function (Figure 1), that manipulates values of a type $\tau_1 = [A \to \mathrm{N}; B \to \mathrm{N}]$. The first branch of drift is exactly abstracted by constraint set $c = \{(r, @A) = (x, @A) + 1, (n, \varepsilon) = (x, @A)\}$ and extended paths $P = \{(r, @A), (x, @A), (n, \varepsilon)\}$, for which we have $\gamma^{\mathrm{NPR}}\left((c, P)\right) = \{[x \mapsto A(k); r \mapsto A(k+1); n \mapsto k] \mid k \in \mathbf{R}\}$.

For numeric abstract domains, it may not necessarily make sense to intersect or unite abstract values that talk about different sets of variables. Nevertheless, for NPRs, it might be the case that two abstract values talk about the same set of variables, but projected on different paths. We hence need to give precise meaning to comparing, intersecting or uniting NPRs with different sets of extended variables.

Definition 3.4 (Inclusion for NPRs)**.** We define the inclusion $\sqsubseteq^{\mathrm{NPR}}$ between two NPRs such that $(c_1, P_1) \sqsubseteq^{\mathrm{NPR}} (c_2, P_2)$ holds iff $(c_1, P_1) \sqsubseteq^{D(\mathcal{S})} \mathrm{Add}_{P_1}\left((c_2, P_2)\right)$ and $P_2 \subseteq P_1$.

Definition 3.5 (Intersection and union for NPRs)**.**

$$(c, P) \cap^{\mathrm{NPR}} (c', P') = \underset{P'}{\mathrm{Add}}(c, P) \cap^{D(\mathcal{S})} \underset{P}{\mathrm{Add}}(c', P')$$
$$(c, P) \cup^{\mathrm{NPR}} (c', P') = \underset{P \setminus P'}{\mathrm{Remove}}(c, P) \cup^{D(\mathcal{S})} \underset{P' \setminus P}{\mathrm{Remove}}(c', P')$$

NPRs can inherit a widening from the underlying numeric abstract domain. Nevertheless, handling recursive algebraic types will require to significantly adapt this widening. We defer the presentation of a widening for NPRs to their adaptation to recursive types, in future work.

A major difficulty when defining an abstract domain for algebraic values is the fact that constructors of a sum type are mutually exclusive. NPRs handle this through the 'well-defined' condition on environment projection when defining concretization. Thus, a single NPR refers to at most one constructor for each value of a sum type. For a more precise analysis of programs with pattern matching, we use finite sets of NPRs, denoting the *disjunction* of their concretizations. For example, the final state of the drift function (Figure 1) is exactly abstracted by the *set* of NPRs $\{\{(r, @A) = (x, @A) + 1\}, \{(r, @B) = (x, @B) - 1\}\}$. This use of disjunctive completion is guided by the program's pattern-matches and limited by the types of variables. We defer the precise presentation of finite disjunctions to a future venue.

4 Soundness and Optimality Results

We have proven the required soundness properties for the operations of the NPR abstract domain, as long as the operations on the chosen numerical domain D are already sound. The proof of theorems 4.1 to 4.4 can be found in [2].

Theorem 4.1 (Pre-order). *The relation $\sqsubseteq^{\mathrm{NPR}}$ is a pre-order: it is reflexive and transitive.*

Theorem 4.2 (Soundness of the pre-order). *The relation $\sqsubseteq^{\mathrm{NPR}}$ is sound,* i.e. *the concretization is monotonic for $\sqsubseteq^{\mathrm{NPR}}$. For any NPRs a_1 and a_2, if $a_1 \sqsubseteq^{\mathrm{NPR}} a_2$, then $\gamma^{\mathrm{NPR}}(a_1) \subseteq \gamma^{\mathrm{NPR}}(a_2)$.*

Theorem 4.3 (Soundness of union and intersection). *The union and intersection over NPRs over-approximate their counterpart operations on sets. For any NPRs a_1 and a_2,*

- $\gamma^{\mathrm{NPR}}(a_1) \cup \gamma^{\mathrm{NPR}}(a_2) \subseteq \gamma^{\mathrm{NPR}}\left(a_1 \cup^{\mathrm{NPR}} a_2\right)$, *and*
- $\gamma^{\mathrm{NPR}}(a_1) \cap \gamma^{\mathrm{NPR}}(a_2) \subseteq \gamma^{\mathrm{NPR}}\left(a_1 \cap^{\mathrm{NPR}} a_2\right)$.

Moreover, we have proved that our definition of abstract intersection does not cause any loss of information: it is optimal, provided that $\cap^{D(\mathcal{S})}$ is already optimal.

Theorem 4.4 (Optimality of intersection). *If $\cap^{D(\mathcal{S})}$ is optimal (*i.e.*, for every two numeric abstract values d_1 and d_2 we have $\gamma^{\mathrm{D}(\mathcal{S})}(d_1) \cap \gamma^{\mathrm{D}(\mathcal{S})}(d_2) = \gamma^{\mathrm{D}(\mathcal{S})}\left(d_1 \cap^{D(\mathcal{S})} d_2\right)$) then $\cap^{\mathrm{NPR}}$ is optimal as well: for any two NPRs a_1 and a_2, we have $\gamma^{\mathrm{NPR}}(a_1) \cap \gamma^{\mathrm{NPR}}(a_2) = \gamma^{\mathrm{NPR}}\left(a_1 \cap^{\mathrm{NPR}} a_2\right)$.*

5 Transfer Functions for Algebraic Types

We now give an overview of transfer functions for operations over algebraic data types. They all involve extended paths.

Following the notation of Miné [13], for any operation s of the analyzed language we denote its abstraction by $\mathbb{S}^{\#}[\![s]\!]$. As explained at the end of Section 3, we work with finite disjunctions of NPRs. For simplicity, however, we only present the different $\mathbb{S}^{\#}[\![s]\!]$ for a single NPR. We consider a language with no mutable objects, hence no aliasing problems.

5.1 Field Access

We write y $\leftarrow$ x.f for the operation that accesses field f of variable x and stores it in variable y. We assume that $x \neq y$. The abstraction for x $\leftarrow$ x.f is easier and is elided due to space constraints. A sound abstraction for this operation is

$$\mathbb{S}^{\#}[\![\mathrm{y} \leftarrow \mathrm{x.f}]\!](c,P) = \mathrm{Remove}_{\mathrm{P|y}}(c,P) \cap^{\mathrm{NPR}} \bigcap^{\mathrm{NPR}}_{(x,.fp)\in P}\{(y,p) = (x,.fp)\}$$

where $\mathrm{P|y} = \{(v,p) \in P \mid v = y\}$ is the subset of P of extended paths starting with y. In this formula, $\mathrm{Remove}_{\mathrm{P|y}}(c,P)$ forgets about the previous value of y and intersecting with the values $\{(y,p) = (x,.fp)\}$ copies what we know about $x.f$ into knowledge about y.

5.2 Product Type Creation

We write y $\leftarrow$ { f1 = x1 ; ... ; fn = xn } for the operation that creates a new record $\{\overline{f_i \to x_i}^{\,i\in\{1,\dots,n\}}\}$ and stores it in y. A sound abstraction of this operation is

$$\mathbb{S}^{\#}[\![\mathrm{y} \leftarrow \{\ \mathrm{f1} = \mathrm{x1}\ ;\ \dots\ ;\ \mathrm{fn} = \mathrm{xn}\ \}]\!](c,P) = \mathrm{Remove}_{\mathrm{P|y}}(c,P) \cap^{\mathrm{NPR}} \bigcap^{\mathrm{NPR}}_{(x_i,p)\in P, i\in I, x_i\neq y}\{(y,.f_ip) = (x_i,p)\}$$

In this formula, $\mathrm{Remove}_{\mathrm{P|y}}(c,P)$ forgets all information about y, and intersecting with the values $\{(y,.f_ip) = (x_i,p)\}$ copies knowledge about x_ip (for $x_i \neq y$) into knowledge about $y.f_ip$. For simplicity, we do not give a more precise abstraction, that keeps information about the x_i equal to y.

5.3 Sum Type Creation

We write y $\leftarrow$ A(x) for the operation that stores the variant $A(x)$ in the variable y. Again, the instruction x $\leftarrow$ A(x) requires a different abstraction but poses no major difficulty, so we concentrate on the case $x \neq y$. A sound abstraction is

$$\mathbb{S}^{\#}[\![\mathrm{y} \leftarrow \mathrm{A(x)}]\!](c,P) = \mathrm{Remove}_{\mathrm{P|y}}(c,P) \cap^{\mathrm{NPR}} \bigcap^{\mathrm{NPR}}_{(x,p)\in P}\{(y,@Ap) = (x,p)\}$$

I.e., we forget what we knew about y, then we copy what we knew about x into knowledge about $y@A$.

5.4 Pattern Matching

We write **match** x **with** A1(y1) $\to$ b1| ... |An(yn) $\to$ bn for pattern matching on the variable x and executing the branch b_i when $x = A_i(y_i)$, *i.e.* destructing a sum type. We call (a,P) the abstract value before the match statement. Again, we only consider the case where $x \neq y_i$. A sound abstraction at the beginning of the i-th branch is given by

$$\mathrm{Remove}_{\mathrm{P|y_i}}(c,P) \cap^{\mathrm{NPR}} \bigcap^{\mathrm{NPR}}_{(x,@A_ip)\in P}\{(y_i,p) = (x,@A_ip)\}$$

The abstract value $\{(y_i,p) = (x,@A_ip)\}$ copies any knowledge about $x@A_i$ to y_i and *simultaneously* asserts that the variable x can only be in the case A_i.

Once the abstract values a_i for each branch i of the pattern have been computed, the abstract union of the values $\mathrm{Remove}_{y_i}\, a_i$, is a sound abstraction of the states that are accessible at the end of the pattern matching construct.

6 Related Work

Our approach is completely parametric with respect to the numeric domain D that one chooses to extend. This is possible, as soon as operations on D for name management from Section 2.3 are available. Those operations are present in the Apron library [7]. Two such domains are the polyhedra [4] and the octagons [12] domains. We presented relational domains as subsets of $V \to \mathbf{R}$, following Miné [12]. This presentation is also exposed as the "level 1" interface of the Apron library [7]. Relational domains can also be presented as subsets of $\mathbf{R}^n$, *i.e.* using dimensions instead of variable names, as done in the "level 0" interface of the Apron library.

Modern static analyzers, such as Astrée [3], handle C records by flattening and C unions following the methodology of Miné [11]. Analyzing algebraic types is a different problem. Union types *à la* C often include a tag field so that

a programmer can discriminate between the cases of a union. The cases of a sum type, in contrast, are pairwise disjoint by design. Moreover, C allows type casts in arbitrary places. This is required for unions to work: to transition from a union type to one of its cases, a cast must be performed. Languages with algebraic types do not need to support arbitrary casts, thanks to the pattern matching construct. With their cast-free semantics, algebraic types are thus easier to analyze than unions. Finally, algebraic types are immutable structures, so keeping track of aliases is not necessary.

Previous work on relational analysis for algebraic data types has been done by Andreescu *et al.* [1] who introduced a domain called *correlations*. Correlations can express *equalities* between parts of different algebraic values, whether those parts are numeric or not. In contrast, the NPR abstract domain can express numeric relations that go beyond equality, but only for the *numeric* parts of different algebraic values. Taking the reduced product of correlations and NPRs would allow to combine the expressiveness of both approaches.

In a different research community, *refinement types* [15–17] have been introduced as a way to make types more expressive, by embedding logical expressions in the syntax of types. In particular, they can express relational properties on scalars. This approach is not based on abstract domains, however: analyzers with refinement types generate logical constraints that are discharged by SMT solvers.

7 Conclusion and Future Work

We have proposed the abstract domain of *numeric path relations*, that denotes relations over structured values that embed scalars. The construction is parameterized over an arbitrary relational domain on scalars, hence the expressiveness of the resulting domain can be adapted. A key ingredient is the notion of *projection path* that points inside some part of a structured value. Handling paths becomes subtle in the presence of sum types, because their cases are mutually exlusive. We have proved the correctness of the operations for the resulting domain, and proved some relative optimality results. We also define transfer functions of operations on algebraic values, showing that our domain can be used to analyze programs handling algebraic data types and scalars.

In future work, we want to implement NPRs to assess the expressiveness of the domain and appreciate its practical scalability. Moreover, we will extend NPRs to handle *recursive* algebraic types—such as lists or trees. This challenging extension will require to define an adapted widening, and will greatly expand the range of programs we can analyze. Finally, combining NPRs with the recent extension of correlations to higher-order functions [14] is yet to be studied.

References

[1] Oana Fabiana Andreescu, Thomas Jensen, Stéphane Lescuyer, and Benoît Montagu. 2019. Inferring frame conditions with static correlation analysis. *PACMPL* 3, POPL (2019), 47:1–47:29.

[2] Santiago Bautista, Thomas Jensen, and Benoît Montagu. 2020. *Combining static correlation analysis with relational numeric domains.* Internship report. ENS Rennes. https://sbautista.fr/internship_report_2020.pdf

[3] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings.* 21–30.

[4] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96.

[5] Alain Deutsch. 2003. Static verification of dynamic properties. In *ACM SIGAda 2003 Conference.*

[6] Jacob M. Howe and Andy King. 2009. Logahedra: A New Weakly Relational Domain. In *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5799)*, Zhiming Liu and Anders P. Ravn (Eds.). Springer, 306–320.

[7] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 661–667.

[8] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages.

[9] Francesco Logozzo and Manuel Fähndrich. 2010. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* 75, 9 (2010), 796–807.

[10] Antoine Miné. 2001. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2053)*, Olivier Danvy and Andrzej Filinski (Eds.). Springer, 155–172.

[11] Antoine Miné. 2006. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (Ottawa, Ontario, Canada) *(LCTES '06)*. ACM, New York, NY, USA, 54–63.

[12] Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100.

[13] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Found. Trends Program. Lang.* 4, 3-4 (2017), 120–372.

[14] Benoît Montagu and Thomas Jensen. 2020. Stable Relations and Abstract Interpretation of Higher-Order Programs. *PACMPL* 4, ICFP, Article 119 (Aug. 2020), 30 pages.

[15] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169.

[16] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13.* ACM Press.

[17] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming - ICFP '14.* ACM Press.